// C++ program to find palindromic substrings of a string

#include<bits/stdc++.h>

using namespace std;

// Returna total number of palindrome substring of

// length greater then equal to 2

int CountPS(char str[], int n)

{

// creat empty 2-D matrix that counts all palindrome

// substring. dp[i][j] stores counts of palindromic

// substrings in st[i..j]

int dp[n][n];

memset(dp, 0, sizeof(dp));

// P[i][j] = true if substring str[i..j] is palindrome,

// else false

bool P[n][n];

memset(P, false , sizeof(P));

// palindrome of single lenght

for (int i= 0; i< n; i++)

P[i][i] = true;

// palindrome of length 2

for (int i=0; i<n-1; i++)

{

if (str[i] == str[i+1])

{

P[i][i+1] = true;

dp[i][i+1] = 1 ;

}

}

// Palindromes of length more then 2. This loop is similar

// to Matrix Chain Multiplication. We start with a gap of

// length 2 and fill DP table in a way that gap between

// starting and ending indexes increases one by one by

// outer loop.

for (int gap=2 ; gap<n; gap++)

{

// Pick starting point for current gap

for (int i=0; i<n-gap; i++)

{

// Set ending point

int j = gap + i;

// If current string is palindrome

if (str[i] == str[j] && P[i+1][j-1] )

P[i][j] = true;

// Add current palindrome substring ( + 1)

// and rest palinrome substring (dp[i][j-1] + dp[i+1][j])

// remove common palinrome substrings (- dp[i+1][j-1])

if (P[i][j] == true)

dp[i][j] = dp[i][j-1] + dp[i+1][j] + 1 - dp[i+1][j-1];

else

dp[i][j] = dp[i][j-1] + dp[i+1][j] - dp[i+1][j-1];

}

}

// return total palindromic substrings

return dp[0][n-1];

}

// Driver program

int main()

{

char str[] = "abaab";

int n = strlen(str);

cout << CountPS(str, n) << endl;

return 0;

|  |
| --- |
| // C++ program to find maximum length equal  // character string with k changes  #include <iostream>  **using** **namespace** std;    // function to find the maximum length of  // substring having character ch  **int** findLen(string& A, **int** n, **int** k, **char** ch)  {  **int** maxlen = 1;  **int** cnt = 0;  **int** l = 0, r = 0;        // traverse the whole string  **while** (r < n) {            /\* if character is not same as ch             increase count \*/  **if** (A[r] != ch)              ++cnt;            /\* While  count > k  traverse the string             again until count becomes less than k             and decrease the count when characters             are not same \*/  **while** (cnt > k) {  **if** (A[l] != ch)                  --cnt;              ++l;          }            /\* length of substring will be rightIndex -             leftIndex + 1. Compare this with the maximum             length and return maximum length \*/          maxlen = max(maxlen, r - l + 1);          ++r;      }  **return** maxlen;  }    // function which returns maximum length of substring  **int** answer(string& A, **int** n, **int** k)  {  **int** maxlen = 1;  **for** (**int** i = 0; i < 26; ++i) {          maxlen = max(maxlen, findLen(A, n, k, i+'A'));          maxlen = max(maxlen, findLen(A, n, k, i+'a'));      }  **return** maxlen;  }    // Driver code  **int** main()  {  **int** n = 5, k = 2;      string A = "ABABA";      cout << "Maximum length = " << answer(A, n, k) << endl;        n = 6, k = 4;      string B = "HHHHHH";      cout << "Maximum length = " << answer(B, n, k) << endl;  **return** 0;  } |

}